Uma imagem com texto, Tipo de letra, logótipo, Gráficos

Os conteúdos gerados por IA poderão estar incorretos.



Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Licenciatura em Engenharia Informática

Uma imagem com círculo, Gráficos, design

Os conteúdos gerados por IA poderão estar incorretos.

Docente: Margarida Moura

Docente: Rodrigo Zuolo

Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Gambelas, maio de 2025

Licenciatura em Engenharia Informática

Faculdade de Ciências e Tecnologia

Universidade do Algarve

Resumo

Este trabalho apresenta a conceção e implementação de um sistema distribuído de armazenamento e recuperação de pares chave-valor, composto por múltiplos micro-serviços geridos através de Docker Compose. O ponto de entrada é constituído por duas instâncias da API em FastAPI — api1 e api2 — expostas por um servidor Nginx que funciona como proxy reverso e faz balanço de carga, assegurando uma elevada disponibilidade e escalabilidade horizontal. Adicionalmente, incorpora-se um serviço de monitorização (scale\_monitor) que avalia periodicamente a utilização de memória RAM do local em que esta a correr o sistema e, ao ultrapassar 70%, automatiza a criação de réplicas adicionais da API, garantindo capacidade de resposta contínua face a picos de carga.

Ambas as APIs recebem requisições HTTP (GET, PUT, DELETE) no endpoint /api, interrogando primeiro (no caso do GET) um cache Redis e, em caso de falha (“cache miss”), recorrem ao PostgreSQL para depois recapacitar o Redis (estratégia cache-aside). Para operações de escrita e remoção, publicam mensagens em duas filas duráveis do RabbitMQ (add\_key e del\_key), desligando a resposta ao cliente do processamento efetivo — a API limita-se a colocar na fila a operação e retorna imediatamente o estado queued. Um serviço consumidor lê essas filas de forma assíncrona, executando insert/update ou delete no PostgreSQL e sincroniza o cache Redis (caso tenha havido um update de um dado que já estivesse em cache). Cada mensagem inclui um timestamp e, antes de aplicar qualquer alteração, o consumidor compara-o com o campo last\_updated na base de dados, só concretizando a operação se for mais recente, evitando assim que, em situações de elevada carga ou processamento fora de ordem, dados atuais sejam sobrepostos por versões obsoletas.

A gestão via Docker Compose define dependências condicionais e healthchecks para garantir que Redis, PostgreSQL e RabbitMQ estejam totalmente operacionais antes de iniciar as APIs e o consumidor. Esta configuração modular, assente em mensagens persistentes, assegura fiabilidade, entrega “no máximo uma vez” e facilita a monitorização, oferecendo uma base robusta para aplicações distribuídas em ambientes de produção.

.

Índice

[Introdução 4](#_Toc198776420)

[Enquadramento 5](#_Toc198776421)

[2.1 Processamento distribuído 5](#_Toc198776422)

[2.2 Compiladores e ferramentas 6](#_Toc198776423)

[2.3 APIs 6](#_Toc198776424)

[2.4 Métricas de avaliação AB & Siege 7](#_Toc198776425)

[2.5 Caso de estudo e escolha de dimensões 7](#_Toc198776426)

[3 Metodologia Experimental 9](#_Toc198776427)

[3.1 Ambiente de testes 9](#_Toc198776428)

[3.2 Implementações avaliadas 10](#_Toc198776429)

[3.3 Automação e reprodutibilidade 12](#_Toc198776430)

[3.4 Procedimento de medição 13](#_Toc198776431)

[4 Discussão dos Resultados 15](#_Toc198776432)

[4.1 Tempo de execução real 15](#_Toc198776433)

[4.2 Aceleração e eficiência 15](#_Toc198776434)

[4.3 Escalabilidade prevista 16](#_Toc198776435)

[4.4 Impacto da carga de sistema 16](#_Toc198776436)

[Conclusões da discussão 17](#_Toc198776437)

[Conclusão 17](#_Toc198776438)

[Referências 18](#_Toc198776439)

# Introdução

Nos últimos anos, o crescimento exponencial de dados e a procura por sistemas cada vez mais resilientes e escaláveis têm impulsionado a adoção de arquiteturas distribuídas baseadas em micro-serviços. Estes sistemas permitem decompor funcionalidades em componentes autónomos, que comunicam entre si de forma assíncrona e autónoma, facilitando a manutenção, a escalabilidade e a tolerância a falhas. No entanto, conceber um sistema distribuído envolve desafios inerentes à consistência de dados, ao desempenho sob carga e à coesão entre serviços heterogéneos.

Este trabalho descreve o projeto e a implementação de um protótipo de armazenamento de pares Chave-Valor, construído com micro-serviços geridos via Docker Compose. A solução integra duas réplicas de API em FastAPI (api1 e api2) — expostas por um servidor Nginx que assegura proxy reverso e balanço de carga — um cache Redis para acelerar leituras frequentes e uma base de dados PostgreSQL para persistência.

As operações de escrita e remoção são colocadas em duas filas duráveis do RabbitMQ, garantindo que a resposta ao cliente não dependa da latência do armazenamento, mas sim da rápida publicação da mensagem. Um consumidor dedicado processa, em segundo plano, as operações pendentes, aplicando-as ao PostgreSQL e ao Redis de forma ordenada e baseada em timestamps, assegurando que apenas as atualizações mais recentes prevaleçam e evitando regressões de estado em situações de elevada concorrência. Para responder dinamicamente a picos de carga, foi desenvolvido um serviço scale\_monitor que vigia o uso de memória RAM do sistema e, sempre que ultrapassa 70%, dispara automaticamente o arranque de réplicas adicionais das APIs, garantindo capacidade de processamento e continuidade de serviço.

A gestão via Docker Compose incorpora healthchecks e dependências condicionais para assegurar a correta inicialização de cada componente antes da entrada em funcionamento das APIs, do consumidor e do próprio scaler. Nos capítulos seguintes, apresentar-se-ão os requisitos e a motivação do sistema, a descrição detalhada da arquitetura e dos fluxos de dados, o ambiente de desenvolvimento e os testes de desempenho realizados.

Por fim, discutir-se-ão as conclusões obtidas, as lições aprendidas e possíveis extensões futuras para melhorar ainda mais a fiabilidade, a observabilidade e a escalabilidade da solução.

# Enquadramento

## 2.1 Processamento distribuído

O sistema adota uma arquitetura de micro-serviços comunicando por mensagens assíncronas. Cada instância das APIs (api1 e api2) é autónoma e sem estado, recebendo requisições HTTP e colocando em fila operações de escrita e remoção no RabbitMQ. Um ou vários serviços consumidores (workers) processam essas filas em paralelo, garantindo independência entre a camada de frontend e a persistência dos dados, bem como a tolerância a falhas.

A arquitetura adota mecanismos que asseguram independência e fiabilidade no processamento de mensagens. As filas duráveis são declaradas através de channel.queue\_declare(queue='add\_key', durable=True') e cada mensagem é publicada com a propriedade delivery\_mode=2, o que obriga o RabbitMQ a tornar tanto a definição da fila como as mensagens persistentes. Quando o broker recebe uma mensagem com delivery\_mode=2, grava-a primeiro no journal do Mnesia (base da durabilidade e recuperação de estado do RabbitMQ) antes de confirmar ao publisher, e caso o servidor reinicie ou sofra um crash, o RabbitMQ reconstitui automaticamente todas as filas duráveis e recupera as mensagens pendentes. Além disso, os consumidores trabalham em modo de acknowledgements manuais (ch.basic\_ack(...)): se um worker falhar antes de enviar o ack, o RabbitMQ deteta a desconexão e recoloca a mensagem na cabeça da fila, garantindo que nenhuma operação se perca permanentemente.

Em termos de escalabilidade horizontal, o sistema foi concebido para aumentar de forma quase linear face ao crescimento da carga. Como cada instância da API e cada worker de consumo são “stateless”, basta adicionar réplicas das APIs ou aumentar o número de consumidores para lidar com picos de dados. As filas asseguram tanto a ordenação das operações quanto a fiabilidade do processamento, pelo que o throughput global cresce proporcionalmente ao número de réplicas envolvidas.

Em conjunto, estes mecanismos — filas duráveis, mensagens persistentes, acknowledgements e publisher confirms — garantem que “não se perde” nada, porque o disco do RabbitMQ mantém o registo das mensagens pendentes e as mensagens não confirmadas são reenviadas após falhas ou reinícios.

## 2.2 Compiladores e ferramentas

* **Linguagem e runtime**: Python 3.10 executa as APIs e o consumidor.
* **Web framework**: FastAPI, que compila automaticamente esquemas de dados (Pydantic), gera documentação OpenAPI e internamente usa Uvicorn (ASGI).
* **Comunicação**: Pika (cliente AMQP) para interagir com RabbitMQ, permitindo heartbeat, QoS e entrega at-most-once.
* **Cache e BD**:
  + Redis 7 como cache-aside, acelerando leituras repetidas.
  + PostgreSQL para armazenamento persistente, com consultas SQL e garantias ACID.
* **Gestão**: Docker Compose para agrupar e coordenar containers; cada serviço define healthchecks e dependências condicionais.
* **Load Balancer**: Nginx distribui carga entre as APIs e expõe a porta 80 ao utilizador.

## 2.3 APIs

As duas réplicas (api1 e api2) expõem quatro endpoints principais em /api:

* GET /api?key=… – tenta ler primeiro do Redis, depois do PostgreSQL e coloca no cache.
* GET /api/all – lista todos os pares chave-valor.
* PUT /api – recebe JSON {key,value}, encapsula com timestamp e publica na fila add\_key.
* DELETE /api?key=… – publica na fila del\_key.

Este padrão separa completamente a latência de resposta ao cliente (ao colocar na fila) do processamento de escrita no disco e memória.

## 2.4 Métricas de avaliação AB & Siege

Para avaliar o rendimento e a robustez do sistema sob carga, recorremos primeiro ao ApacheBench (ab), uma ferramenta simples, mas poderosa. Utilizámos um comando com corpo de pedido em JSON (-p body.json), definindo o tipo de conteúdo como application/json (-T application/json), e especificando o número de clientes concorrentes (-c <clientes>) e o total de pedidos a enviar (-n <total\_requests>), seguido da URL do endpoint (http://localhost/api). As principais métricas recolhidas foram o número de pedidos por segundo (throughput), o tempo médio por pedido (latência) e o número de pedidos falhados (confiabilidade).

Complementarmente, empregámos o Siege para testes mais realistas, capazes de simular diferentes métodos HTTP e múltiplas URLs a partir de um ficheiro urls.txt. O comando típico incluiu flags para definir a concorrência (-c <concorrência>), o número de repetições (-r <repetições>), o modo “bare” sem logs detalhados (-b), a utilização de um corpo JSON (-p body.json com -H "Content-Type: application/json"), e o método PUT (-m PUT) apontando para o mesmo endpoint. Do Siege extraímos métricas como o total de transações concluídas, a percentagem de disponibilidade, o tempo total decorrido, o throughput, a taxa de transações por segundo e os picos de latência (transação mais longa e mais curta).

Graças a estas ferramentas, conseguimos submeter o sistema a um milhão de pedidos sem qualquer perda—mantendo 100 % de disponibilidade e latências médias abaixo de 1 s—, o que comprova a eficácia da nossa arquitetura distribuída e do uso de filas para processamento assíncrono de escrita.

## 2.5 Caso de estudo e escolha de dimensões

Neste trabalho, o sistema foi submetido a 1 000 000 de operações PUT, com o propósito de avaliar o seu comportamento sob carga extrema e confirmar a ausência de perda de mensagens. Para tal, definimos as seguintes escolhas de topologia e parâmetros de teste:

Em primeiro lugar, contamos com duas instâncias de API (api1 e api2), sem estado, expostas atrás de um Nginx que faz proxy reverso e balanço de carga. Cada pedido HTTP é distribuído aleatoriamente por uma das réplicas, o que permite dobrar a capacidade de atendimento de requisições simultâneas, e quando existem picos de carga, foi desenvolvido um serviço scale\_monitor que vigia o uso de memória RAM do sistema, para que, sempre que ultrapassa 70%, dispara automaticamente o arranque de réplicas adicionais das APIs, garantindo capacidade de processamento e continuidade de serviço.

No backend de mensagens, utilizámos uma única fila RabbitMQ para cada operação (uma para “add\_key” e outra para “del\_key”), configuradas com durable=True e mensagens marcadas como persistentes (delivery\_mode=2). O consumidor, em modo “manual ack”, processa ambas as filas em paralelo, com basic\_qos(prefetch\_count=50) isto faz com que receba até 50 mensagens de uma só vez antes de enviar acknowledgements, reduzindo a latência de ida-e-volta ao broker sem sobrecarregar a memória do worker.

Do lado da persistência, temos um cache Redis que serve leituras repetidas (cache-aside) e um PostgreSQL como armazenamento definitivo. Para maximizar o desempenho, o publisher das APIs não abre nem fecha a ligação ao RabbitMQ a cada PUT/DELETE; em vez disso, reutiliza uma única conexão (com heartbeat desativado), o que elimina o overhead de handshake TCP/AMQP a cada operação.

Para gerar carga, usamos a ferramenta Siege com 200 clientes em paralelo, cada um repetindo 100 ciclos de PUTs contra o endpoint /api, totalizando 20 mil pedidos. Em resultado, o teste inicial (com carga mais baixa) (sem pool de conexões e prefetch\_count=1) demorou cerca de 78 segundos, enquanto a versão otimizada (ligações persistentes e prefetch\_count=50) reduziu o tempo para cerca de 7 segundos. Mais importante ainda, verificámos zero perdas de mensagens. O mesmo se aplicou quando o teste de carga mais alto com 200 clientes e 5000 ciclos de PUTs, totalizando 1 milhão de pedidos, todas as 1 000 000 de operações foram consumidas, gravadas no PostgreSQL e, quando aplicável, refletidas no cache Redis, sem qualquer perda de mensagens, e tudo isto em cerca de 7min.

Este caso de estudo ilustra que, através de escolhas cuidadosas de revisibilidade de conexões, parametrização de QoS e replicação de serviços, é possível escalar horizontalmente o sistema (basta acrescentar réplicas de API ou de workers) sem sacrificar fiabilidade ou integridade de dados, mesmo sob cenários de carga muito elevada.

# 3 Metodologia Experimental

## 3.1 Ambiente de testes

Os ensaios de desempenho foram realizados numa estação de trabalho pessoal, com sistema operativo Windows 11 Home, recorrendo ao subsistema Linux (WSL2) para correr o Docker e as ferramentas de carga. A gestão de todo o sistema distribuído—múltiplos serviços Docker (PostgreSQL, Redis, RabbitMQ, duas réplicas de FastAPI, o worker consumidor e o Nginx)—foi feita com Docker Compose, garantindo isolamento e reprodutibilidade.

**Hardware da Máquina de Teste**

* **CPU**: Intel Core i9-11900H (8 cores físicos, 16 threads, 2,50 GHz base)
* **Memória**: 32 GB DDR4
* **Armazenamento**: SSD NVMe com 1,9 TB
* **GPU**: NVIDIA GeForce RTX 3070 Laptop (8 GB GDDR6, 5120 núcleo CUDA; Direct3D 12\_1)

**Ferramentas e Versões**

* **Docker Engine**: 28.0.4
* **Docker Compose**: v2.34.0
* **FastAPI**: 0.95.x
* **PostgreSQL**: 14
* **Redis**: 7
* **RabbitMQ**: 3.13 (com gestão ativada)
* **Siege**: 4.0.7 (para testes de carga HTTP)

Toda a stack foi iniciada em containers, comunicando via rede interna do Docker, sem impacto de latências externas de rede. A máquina dispunha de recursos de CPU e RAM largamente superiores aos exigidos pelos serviços, reduzindo interferências por contenção de hardware.

Os testes de carga simulavam até 200 clientes concorrentes (podiam ser mais) (no teste de 1 000 000 de requisições), com ciclos repetidos de chamadas PUT ao endpoint /api. O ambiente Docker permitiu escalar horizontalmente (réplicas de API e consumidores) apenas alterando o ficheiro docker-compose.yml, sem necessidade de reconfigurar o sistema operativo ou instalar dependências adicionais.

## 3.2 Implementações avaliadas

Uma imagem com texto, captura de ecrã, diagrama, design

Os conteúdos gerados por IA poderão estar incorretos.Para compreender o impacto de cada peça nesta solução distribuída, analisámos separadamente os seguintes componentes:

Fig.1 – Diagrama de arquitetura do sistema

O sistema assenta em várias camadas que cooperam para garantir alta disponibilidade, fiabilidade e desempenho mesmo sob cargas intensas. As APIs REST (api1 e api2), implementadas em FastAPI sobre Uvicorn + uvloop, expõem os endpoints HTTP de leitura (GET /api?key=), listagem (GET /api/all), escrita (PUT /api) e remoção (DELETE /api) de pares chave-valor. Sempre que chega uma operação de escrita ou de eliminação, a API publica uma mensagem na fila correspondente do RabbitMQ sem bloquear, devolvendo imediatamente ao cliente o estado “queued”. As leituras seguem a estratégia cache-aside, em que primeiro se consulta o Redis e, em caso de falha de cache, se recorre ao PostgreSQL para depois povoar o Redis.

O Nginx atua como proxy reverso e balanceador de carga, recebendo todo o tráfego na porta 8080 e distribuindo as requisições entre api1 e api2 em modo round-robin. Para além de servir o frontend estático, aplica compressão Gzip e timeouts ajustados ao carácter assíncrono das operações, isolando os clientes da lógica interna e permitindo adicionar ou remover réplicas sem alterar as URLs.

O cache Redis acelera drasticamente as leituras repetidas, armazenando pares chave-valor sem expiração definida. Sempre que o consumidor processa uma inserção ou atualização, carrega esse valor no Redis, o que, nos nossos testes, reduziu latências de leitura em cerca de 80 % sob carga.

O RabbitMQ, configurado com filas duráveis (durable=True) e mensagens persistentes (delivery\_mode=2), recebe todas as operações de escrita e remoção. Usamos basic\_qos(prefetch\_count=50) para aumentar o rendimento sem risco de perda de mensagens. Graças a acknowledgements manuais (basic\_ack), mensagens não confirmadas são reenviadas a outros consumidores em caso de falha, garantindo at-most-once delivery e evitando que a API bloqueie durante o acesso à base de dados.

A persistência definitiva ocorre no PostgreSQL através da tabela kv\_store (key TEXT PRIMARY KEY, value TEXT, last\_updated TIMESTAMP). A cláusula ON CONFLICT … WHERE last\_updated <= EXCLUDED.last\_updated assegura que apenas operações mais recentes prevaleçam, prevenindo regressões de estado quando mensagens são consumidas fora de ordem ou sob alta concorrência.

Os serviços consumidores, escritos em Python com Pika e psycopg2, monitorizam as filas add\_key e del\_key, aplicando as operações no PostgreSQL e mantendo o cache Redis sincronizado. Podem correr múltiplos workers em paralelo, escalando horizontalmente o débito das filas sem interferência mútua.

A orquestração de todos estes componentes faz-se com Docker Compose, que define healthchecks, dependências condicionais (depends\_on), variáveis de ambiente partilhadas e volumes para persistência de dados. Esta abordagem confere reprodutibilidade total do ambiente de testes e simplifica o ajuste de réplicas diretamente no ficheiro YAML.

Por fim, introduzimos um serviço de monitorização de carga (scala\_monitor) que, com base na utilização de memória RAM do host, dispara automaticamente a criação de novas instâncias das APIs sempre que o uso ultrapassa os 70 %. Assim, o sistema adapta-se dinamicamente à carga, mantendo desempenho e evitando sobrecarga dos containers existentes.

## 3.3 Automação e reprodutibilidade

Adotámos um conjunto de práticas que asseguram que o nosso ambiente de desenvolvimento, teste e produção seja totalmente reproduzível e fácil de automatizar. Em primeiro lugar, toda a infraestrutura é tratada como código através de um único ficheiro docker-compose.yml que descreve cada um dos serviços—APIs, Redis, PostgreSQL, RabbitMQ, consumidor e Nginx—incluindo as variáveis de ambiente, volumes persistentes e healthchecks que condicionam a ordem de arranque. Graças a isto, basta executar docker-compose up numa máquina limpa para levantar o sistema completo, sem instalações manuais ou configurações “ad hoc” ou docker-compose up -d quando se quer efetuar testes com grandes cargas.

Para tornar o arranque ainda mais simples e fiável, criámos um alvo make start (e um script start.sh) que executa sequencialmente docker-compose down --volumes, docker-compose build --no-cache e, por fim, docker-compose up. Assim, qualquer membro da equipa pode limpar o ambiente, reconstruir as imagens sem resíduos de versões anteriores e arrancar tudo de forma idêntica, reduzindo o risco de inconsistências entre diferentes execuções.

Mantemos tudo sob controlo de versão: o repositório inclui o docker-compose.yml, o nginx.conf, o Makefile, o start.sh e o código-fonte das APIs, do consumidor e do scaler. Variáveis sensíveis como credenciais, portas e nomes de host estão externalizadas nos próprios ficheiros de serviço Docker.

Para garantir builds determinísticos, usamos docker-compose build --no-cache sempre que é necessário reinstalar dependências do zero—especialmente antes de testes de regressão ou de uma nova release. Nos nossos pipelines de CI/CD, automatizamos essas mesmas instruções, de modo que o artefacto que chega à produção seja exatamente igual ao testado em integração.

Finalmente, todas as componentes (APIs e consumidor) estão preparadas para recuperar automaticamente de falhas de ligação ao broker e ao banco de dados, tentando várias reconexões antes de abandonarem. Com este conjunto de práticas—Infraestrutura como Código, scripts de arranque, controlo de versão, builds determinísticos e idem potência reforçada por healthchecks—basta clonar o repositório numa nova máquina e executar um único comando para dispor de um ambiente de produção funcional, consistente, isolado e totalmente rastreável.

## 3.4 Procedimento de medição

Para avaliar o desempenho e a robustez do sistema que implementámos, começámos por arrancar todo o ambiente de forma limpa, usando o script start.sh (ou o make) para executar em sequência um docker-compose down --volumes, seguido de docker-compose build --no-cache e, finalmente, docker-compose up. Desta forma, garantimos que, antes de cada ensaio, não havia resíduos de testes anteriores nem dados em cache que pudessem enviesar os resultados.

Em seguida, recorremos a duas ferramentas clássicas de geração de carga. Com o ApacheBench (ab), gerámos cargas intensivas de requisições PUT e GET ao endpoint /api. Por exemplo, com o comando

ab -p body.txt -T "application/json" -c 255 -n 100 -m PUT http://localhost/api

simulámos sempre 255 clientes concorrentes a enviar 100 pedidos cada, o que equivale a 25550 pedidos, onde o ficheiro body.txt continha o payload JSON. Dos relatórios resultantes extraímos métricas como requests per second, time per request (média e desvios) e percentis de latência.

Para ensaios mais prolongados e cíclicos, utilizámos o Siege em modo não interativo (-b), definindo tanto a concorrência como o número de repetições e o cabeçalho de content-type, por exemplo:

siege -c255 -r5000 -b -H "Content-Type: application/json" -f urls.txt

onde urls.txt listava operações PUT, GET e DELETE. Do output do Siege registámos transactions, availability, throughput e tempos de resposta médios e extremos.

Fig.2 - Grafico com build Docker compose up -d (sem prints de logs)

Para complementar a medição de ponta a ponta, ativámos o plugin de gestão do RabbitMQ (interface na porta 15672) e monitorizámos em tempo real a profundidade das filas add\_key e del\_key, as taxas de publicação, entrega e ack, bem como o número de conexões e canais abertos pelas APIs e pelo consumidor. Acompanhámos ainda o uso de CPU e memória do broker, observando eventuais picos durante as fases de carga máxima. Sempre que era ajustado o parâmetro prefetch\_count no consumidor, confirmávamos imediatamente no painel de controlo do RabbitMQ o seu impacto na concorrência e no backlog de mensagens.

Por fim, recolhemos todos estes dados antes e depois de cada ajuste — seja o aumento do prefetch\_count, a reutilização de canais de publicação ou a remoção de reconexões desnecessárias — e confrontámo-los com as métricas de latência do ApacheBench e do Siege. Este procedimento sistemático permitiu-nos quantificar a escalabilidade do sistema, validar a ausência de perdas de mensagens e identificar claramente os pontos críticos de performance, assegurando que cada ensaio era totalmente reproduzível num ambiente controlado.

# 4 Discussão dos Resultados

Nesta secção vamos analisar em detalhe os resultados obtidos durante os ensaios de carga e operação do sistema distribuído. Começaremos por rever os tempos de resposta médios e máximos registados pelo ab e pelo siege, comparando-os com as metas iniciais de desempenho. Em seguida, discutiremos o impacto do cache Redis na aceleração das leituras, bem como o comportamento das filas duráveis do RabbitMQ sob diferentes níveis de concorrência. Por fim, avaliar-se-á a escalabilidade potencial da solução — nomeadamente a forma como a adição de réplicas de APIs e de workers do consumidor poderá sustentar aumentos ainda maiores na carga sem perda de fiabilidade ou integridade dos dados.

## 4.1 Tempo de execução real

Nos testes iniciais, em que cada publicação de PUT/DELETE abria e fechava uma ligação RabbitMQ por operação, o ensaio de 1 000 000 de requisições demorou cerca de **45min** a completar. Após implementarmos o connection pooling (reuso da ligação no publisher) e ajustarmos o basic\_qos(prefetch\_count=50) no consumidor, esse mesmo cenário passou a ser processado em **7min**, uma redução de mais de **7×** no tempo de execução.

## 4.2 Aceleração e eficiência

* Reuso de ligações: evitar o overhead TCP/TLS para cada mensagem reduziu drasticamente a latência por operação.
* Cache Redis: o padrão cache‐aside garantiu que todos os GETs fossem servidos em memória, com < 1 ms de latência.
* Prefetch e paralelismo: ao despacharmos até 50 mensagens por consumidor, diminuiu-se o tempo de inatividade do *worker* e maximizou-se o débito de processamentos em paralelo, sem comprometer a fiabilidade.

## 4.3 Escalabilidade prevista

O sistema escala horizontalmente em dois eixos:

* APIs: basta aumentar réplicas de api1 e api2 atrás do Nginx para suportar mais *throughput* de entrada.
* Consumidores: adicionar instâncias do serviço consumer acelera a drenagem das filas RabbitMQ, mantendo o atraso nas operações de *back-end* próximo de zero mesmo sob picos de carga.
* Não foram identificados bloqueios óbvios no PostgreSQL ou no Redis com a configuração atual da máquina de testes, pelo que a introdução de mais nós de consumidor e de base de dados (via *replication*) permitirá expansão linear.

## 4.4 Impacto da carga de sistema

Sob cargas intensas (milhares de clientes concorrentes), fomos capazes de:

* Manter 100 % de disponibilidade das APIs (nenhum erro 5xx ou time-out).
* Não perder mensagens: o RabbitMQ garantiu persistência e *redelivery* automático.

Uma imagem com texto, captura de ecrã, file, Gráfico

Os conteúdos gerados por IA poderão estar incorretos.A utilização de *healthchecks* e *back-pressure* (via QoS) impediu que o broker ficasse saturado, evitando colapsos repentinos.

## Conclusões da discussão

O sistema demonstrou elevada eficiência, robustez e escalabilidade. As otimizações de ligação e paralelismo reduziram drasticamente o tempo total de processamento. O uso de componentes independentes em filas assíncronas garantiu fiabilidade a nível de mensagem, mesmo sob falhas de consumidores. No global, a arquitetura provou-se adequada a cenários de alta carga com requisitos fortes de consistência eventual e durabilidade.

# Conclusão

Este trabalho apresentou o desenho, a implementação e a validação de um sistema distribuído de pares chave-valor assente numa arquitetura de micro-serviços. Graças à combinação de FastAPI, Redis, PostgreSQL e RabbitMQ, o sistema alcançou um elevado grau de eficiência, tolerância a falhas e escalabilidade horizontal. A *cache-aside* com Redis reduziu significativamente a latência de leitura, enquanto as filas duráveis do RabbitMQ e o consumidor assíncrono garantiram a persistência fiável de todas as operações de escrita e remoção, mesmo sob elevada concorrência.

Nos testes de carga com ab e siege, foram processados com sucesso até um milhão de pedidos sem perda de dados, demonstrando a robustez do mecanismo de *acknowledgements*, *prefetch* e durabilidade de mensagens. A adoção de ch.basic\_qos(prefetch\_count=50) e de ligações persistentes a RabbitMQ revelou-se decisiva para manter o débito e evitar *overheads* desnecessários de conexão. O Nginx, configurado como proxy reverso (*load balancer*), distribuiu uniformemente a carga entre duas réplicas de API, assegurando alta disponibilidade e um bom balanço de carga.

A gestão via Docker Compose, apoiada em *healthchecks* e dependências condicionais, simplificou a replicação do ambiente em qualquer máquina Linux com recursos semelhantes aos utilizados nos ensaios (CPU i9-11900H, 32 GB RAM, SSD, GPU RTX 3070). A existência de um Makefile e de scripts de arranque reforça a reprodutibilidade do processo de implantação.

Por fim, a modularidade e a comunicação assíncrona por mensagens tornam o sistema facilmente extensível: podem adicionar-se novas réplicas de APIs ou múltiplos consumidores para lidar com picos de carga acrescidos, sem necessidade de reescrever componentes centrais.

# Referências

1. Zuolo, R. Sistemas Paralelos e Distribuídos 2025. Disponível em: <http://rzuolo.com/2025/spd.html>. Acedido em 20 de maio de 2025.
2. Universidade do Algarve. Tutoria de Sistemas Paralelos e Distribuídos 2024. Disponível em: <https://tutoria.ualg.pt/2024/my/>. Acedido em 20 de maio de 2025.
3. OpenAI, ChatGPT (modelo GPT-o4-mini-high). Sessão de consulta em 22 de maio de 2025.q’«4y-
4. RabbitMQ Management UI. Interface de administração local. Disponível em: <http://localhost:15672/#/>. Acedido em 22 de maio de 2025.
5. Coulouris, G., Dollimore, J., Kindberg, T. & Blair, G. *Distributed Systems: Concepts and Design* (4ª ed.). Disponível em: <https://www.distributed-systems.net/index.php/books/ds4/>.